Colas de prioridad – *Heaps* Binarios

Ingeniería en Computación e Informática

Tabla de contenidos

Introducción

- Una cola de prioridad es un tipo de datos abstracto que almacena un conjunto de datos que poseen una llave perteneciente a algún conjunto ordenado.
- Permite insertar nuevos elementos y extraer el máximo (o el mínimo, en caso de que la estructura se organice con un criterio de orden inverso).
- Numerosas aplicaciones: Sistemas operativos, algoritmos de scheduling, gestión de colas en cualquier ambiente, etc.

Introducción

- La implementación más eficiente es a través de *heap*.
- *Heap* significa, literalmente, “montón”.
- La principal característica e importancia del *Heap* es que se puede obtener el mínimo elemento en tiempo constante $O(1)$.

Heap

Características

1. Propiedad estructural:
   - Es un Árbol Binario Completo. Propiedades:
     - Su altura es a lo sumo $(\log n)$.
     - Admite una representación implícita sobre un arreglo.
2. Propiedad de orden:
   - El valor de cualquier nodo es menor o igual que el de sus hijos (Min-*Heap*).
   - De manera análoga se define un Max-*heap*.

Heap

Características

- Un heap es un AB completo con la propiedad de orden enunciada.
- Propiedades:
  - Árbol binario perfectamente balanceado.
  - Todas las ramas del árbol son secuencias ordenadas.
  - La raíz del árbol es el nodo de valor mínimo (o máximo en un Max-*Heap*).
  - Todo subárbol de un Heap es también un *Heap*.
  - (no obligatorio): es "izquierdista", o sea, el último nivel está lleno desde la izquierda.
  - (Ojo: ¡no es un ABB, ni una estructura totalmente ordenada!)

Heap
Representación

- Todas las representaciones usadas para árboles binarios son admisibles.
- Representación con punteros, eventualmente con punteros hijo-padre.
- Representación con arreglos: particularmente eficiente.

Heap

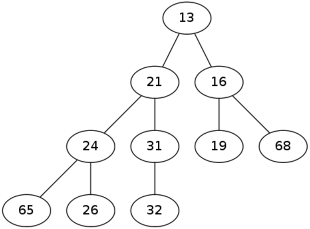
Representación : Arreglos

- Ventajas:
  - Muy eficientes en términos de espacio (¡ver desventajas!).
  - Facilidad de navegación.
- Desventaja:
  - Implementación estática (puede ser necesario duplicar el arreglo (o achicarlo) a medida que se agregan/eliminan elementos.

Heap

Representación : Arreglos

- $i =$ posición en el arreglo.
- Hijo Izquierdo $= 2i$.
- Hijo Derecho $= 2i + 1$.
- Padre $=$ (int)$i/2$.



13	21	16	24	31	19	68	65	26	32
1	2	3	4	5	6	7	8	9	10

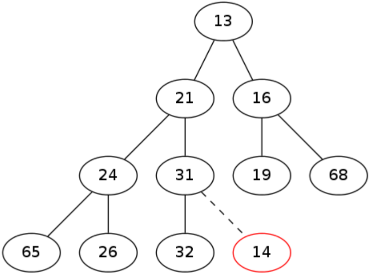
Operaciones del Heap

Insertar

- Insertar en la posición correspondiente para que siga siendo un árbol completo, es decir, al final de izquierda a derecha.
- Reorganizar para que cumpla las condiciones del *heap*, haciéndolo flotar:
  - Comparar con el nodo padre: si no cumple las condiciones del árbol mínimo/máximo, entonces intercambiar ambos.

Operaciones del Heap

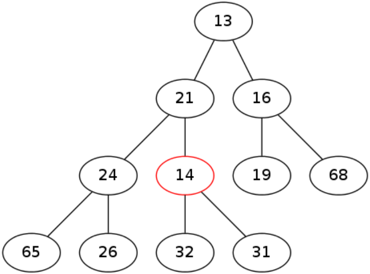
Insertar : Ejemplo



13	21	16	24	31	19	68	65	26	32	14
1	2	3	4	5	6	7	8	9	10	11

Operaciones del Heap

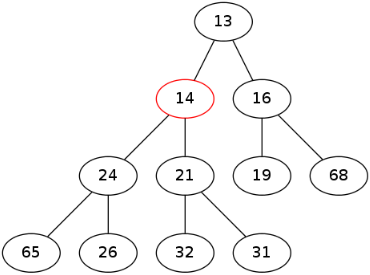
Insertar : Ejemplo



13	21	16	24	14	19	68	65	26	32	31
1	2	3	4	5	6	7	8	9	10	11

Operaciones del Heap

Insertar : Ejemplo



13	14	16	24	21	19	68	65	26	32	31
1	2	3	4	5	6	7	8	9	10	11

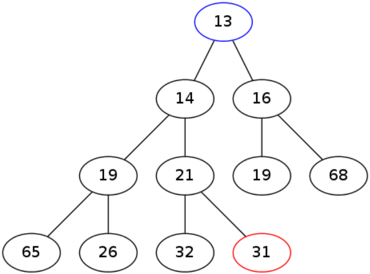
Operaciones del Heap

Eliminar

- Se elimina mínimo/máximo elemento. Éste está en la posición 1 de arreglo y raíz del árbol.
- El último nodo lo colocaremos en la posición uno, y lo *hundiremos*
- Se hunde en la dirección del hijo menor para mantener la propiedad de orden.

Operaciones del Heap

Eliminar : Ejemplo

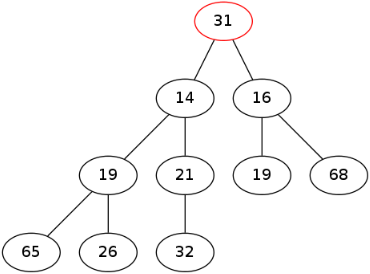


13	14	16	19	21	19	68	65	26	32	31
1	2	3	4	5	6	7	8	9	10	11

Universidad Andrés Bello

Operaciones del Heap

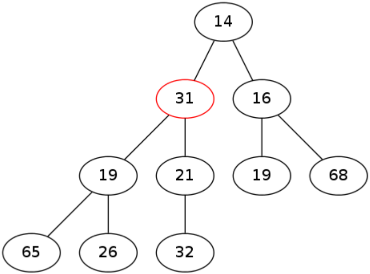
Eliminar : Ejemplo



31	14	16	19	21	19	68	65	26	32	
1	2	3	4	5	6	7	8	9	10	11

Operaciones del Heap

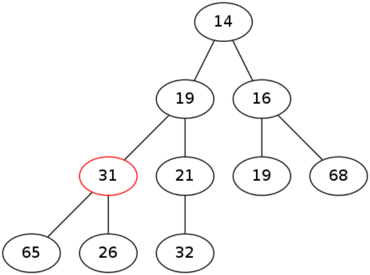
Eliminar : Ejemplo



14	31	16	19	21	19	68	65	26	32	
1	2	3	4	5	6	7	8	9	10	11

Operaciones del Heap

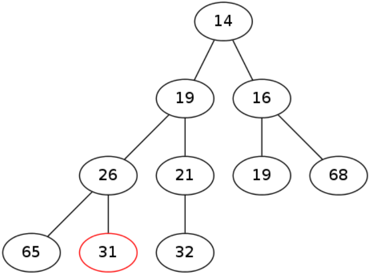
Eliminar : Ejemplo



14	19	16	31	21	19	68	65	26	32	
1	2	3	4	5	6	7	8	9	10	11

Operaciones del Heap

Eliminar : Ejemplo



14	19	16	26	21	19	68	65	31	32	
1	2	3	4	5	6	7	8	9	10	11

Operaciones del Heap

Eficiencia de las Operaciones

Operaciones	Arreglo Estático	Arreglo Dinámico	Lista Simplemente Enlazada	Lista Doblemente Enlazada	Árbol Binario de Búsqueda	*Heap* Binario
Acceder	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$	–
Buscar	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
Insertar	–	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
Eliminar	–	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$

Otros Heaps

Eficiencia

Heap	Buscar Mín/Máx	Extraer Mín/Máx	Dism./Aum. Clave	Insertar	Eliminar	Mezclar
Binario	$O(1)$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(m+n)$
Binomial	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Fibonacci	$O(1)$	$O(\log(n))$*	$O(1)$*	$O(1)$	$O(\log(n))$*	$O(1)$

(*)Tiempo Amortizado.

Heapsort

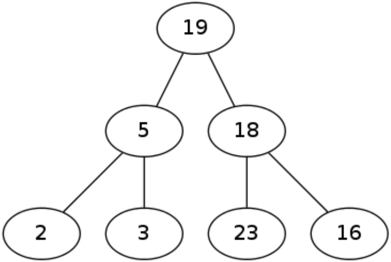
Funcionamiento

- A partir de un arreglo desordenado que se quiere ordenar ascendentemente:
  - Para $i = n/2$ hasta 1:
    - Hundir i.
  - Para $i = n$ hasta 2:
    - Intercambiar posición 1 con i.
    - Hundir i.

Heapsort

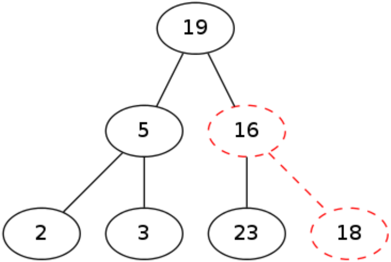
Ejemplo

1	2	3	4	5	6	7
19	5	18	2	3	23	16



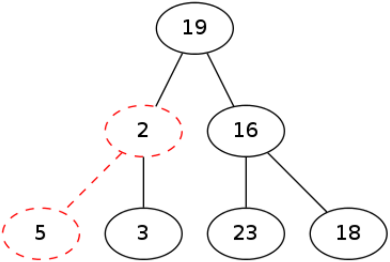
Heapsort
Ejemplo

- Hundir $i = 3$.



Heapsort
Ejemplo

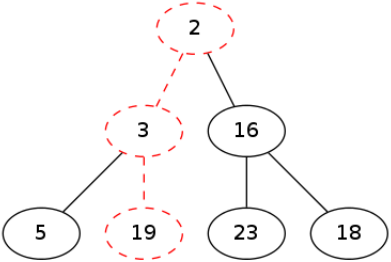
- Hundir $i = 2$.



Heapsort

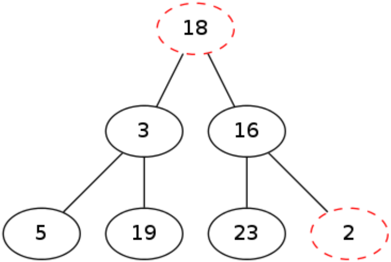
Ejemplo

- Hundir $i = 1$.

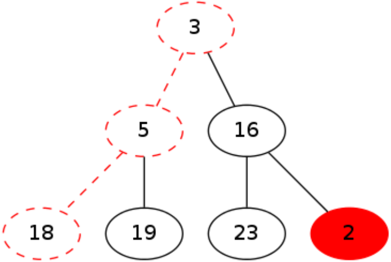


Heapsort
Ejemplo

- Intercambiar 1 con 7.



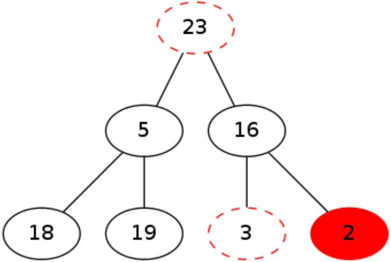
- Hundir raíz ($i = 0$).



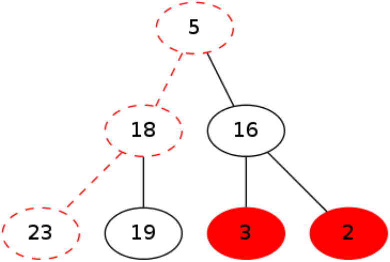
Heapsort

Ejemplo

- Intercambiar 1 con 6.

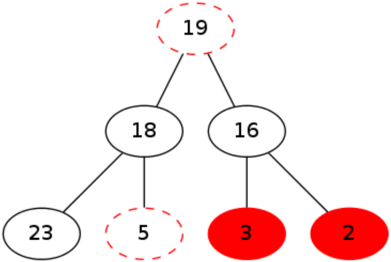


- Hundir raíz ($i = 0$).

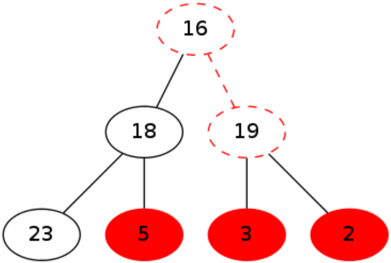


Heapsort
Ejemplo

- Intercambiar 1 con 5.



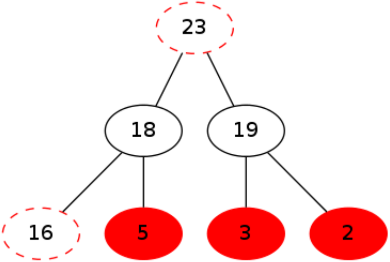
- Hundir raíz ($i = 0$).



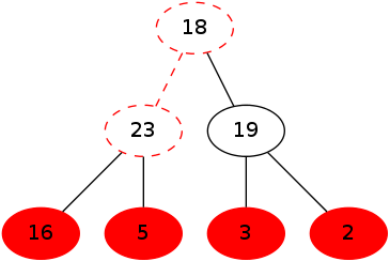
Heapsort

Ejemplo

- Intercambiar 1 con 4.

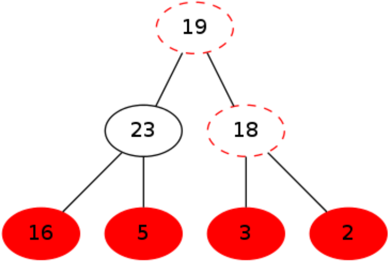


- Hundir raíz ($i = 0$).

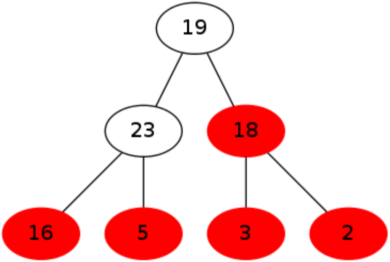


Heapsort
Ejemplo

- Intercambiar 1 con 3.

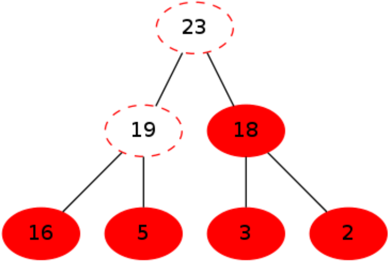


- Hundir raíz $(i = 0)$.

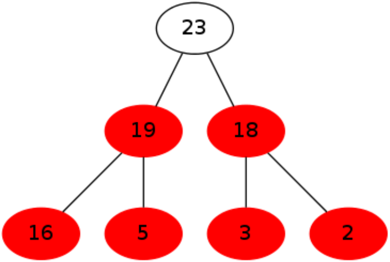


Heapsort
Ejemplo

- Intercambiar 1 con 2.

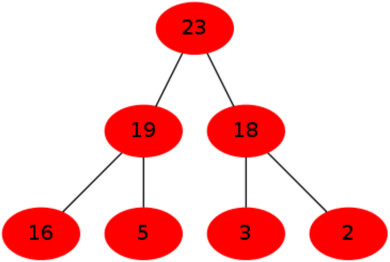


- Hundir raíz ($i = 0$).



Heapsort

Ejemplo



Heapsort

Eficiencia

- La eficiencia para un algoritmo Heapsort está dominado por el ordenamiento hacia abajo $O(n \log n)$ y no por la construcción $O(n)$.
- Heapsort garantiza ordenar n elementos en su lugar en un tiempo proporcional a $n \log n$ sin importar la entrada.
- No hay entrada de peor caso que haga el algoritmos Heapsort significativamente más lento (como es el caso de Quicksort).

Heapsort

Eficiencia

- No utiliza espacio adicional (como es el caso de Mergesort).
- El ciclo interno (costo por comparación) tiene más operaciones básicas que Quicksort y utiliza más comparaciones que Quicksort para entradas aleatorias.
- Heapsort también es útil para problemas de selección, como encontrar el k elemento mayor entre n elementos (deteniendo el algoritmo después de k extracciones).